Software Security Issues in Embedded Systems

Somesh Jha
Computer Sciences Department,
University of Wisconsin, Madison, WI 53706.

1 Introduction

Embedded systems and networks are becoming increasingly prevalent in critical sectors, such as medical and defense sectors. Therefore, malicious or accidental failures in embedded systems can have dire consequences. Hence, the integrity of embedded software infrastructures, such as configuration and code, is of paramount importance. The autonomous nature of embedded systems also poses new challenges in the context of system integrity. Since embedded systems are reactive, unexpected or malicious environment events or can cause failures, which can have dire consequences in critical sectors. Embedded systems and networks also often have to operate autonomously in a dynamic environment. Therefore, an embedded system has to adapt its behavior to the change in environment or the overall goal. Unauthorized or unverified updates to the infrastructure of an embedded system can also compromise its integrity.

In recent years, there have been significant advances in the area of software security. There have been various techniques developed in the context of software security, such as automated signature generation, vulnerability assessment, and detecting malicious behavior. However, all these techniques are not directly applicable in the context of embedded systems because of following reasons:

- *Dynamic and configurable environment:* Embedded systems are generally deployed in environments that are highly dynamic and configurable. For example, the environment of an embedded system deployed in a battlefield is extremely dynamic.
- *Changing functional requirements:* Functional requirements of an embedded system change over time. Functional requirements of an embedded system deployed in a battleship change with mission of the operation.
- *Interconnected network of components:* Frequently an embedded system is a complex network of components. Therefore, a malicious or accidental fault in a component can lead to a complex cascade of events in the network.
- *Recovery is paramount:* Generally, techniques developed in the realm of software security focus on detection and prevention. Embedded systems are frequently deployed in mission critical applications where consequences of failures can be dire. Therefore, recovery from failures is extremely important in the context of embedded systems.

2 Promising Research Directions

Extending existing techniques in software security to handle the four abovementioned characteristics of embedded systems is an important research direction. I will provide details of two such research directions.

- *Vulnerability assessment and prevention in presence of a dynamic environment:* Existing techniques for vulnerability assessment have been developed for systems (such as servers) whose environments are relatively static. Extending dynamic and static analysis techniques for vulnerability assessment and prevention for systems with dynamic environments is a very interesting research direction. I envision that existing techniques will have to be extended to incorporate specification of the environment. In this context, an interesting research direction would be to generate vulnerability signatures [2, 6] for systems with dynamic environments. I envision the signatures in this case will be parametrized by a specification of the environment, i.e., signatures will only be valid if certain environment conditions are satisfied.

- *Recovery from malicious or accidental faults:* As mentioned before an embedded system is a complex network of components. Therefore, a fault in a component can create a ripple of events throughout the network. This makes recovery for embedded systems extremely challenging. A causality graph for an embedded system is a graph where the nodes are events and edges are the causality between events ($e \rightarrow e'$ means that event e can cause event e'). Techniques for discovering a causality graph of an embedded is essentially for recovering from faults. Essentially the effects of a fault can be determined from examining the causality graph. Techniques for constructing attack graphs [1, 5] and alert correlation [3, 4]

References

[1] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *ACM Conference on Computer and Communications Security*, 2002.

[2] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability based signatures. In *IEEE Symposium on Security and Privacy*, pages 21–24, May 2006.

[3] F. Cuppens and A. Mige. Alert correlation in a cooperative intrusion detection framework. In *IEEE Symposium on Security and Privacy*, 2002.

[4] P. Ning, Y. Cui, and D. S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *ACM Conference on Computer and Communications Security*, 2002.

[5] O. Sheyner, J. W. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, 2002.

[6] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *In the Proceedings of ACM SIGCOMM*, Portland, OR, August 2004.

Software Security Issues in Embedded Systems

Somesh Jha
University of Wisconsin

Software Security

- Vulnerability Assessment
  - Analysis tools for discovering vulnerabilities in source code and binaries
- Automated Signature Generation
  - Generating signatures that filter our malicious inputs
- Malicious Code Detection
  - Detecting whether a binary has malicious behavior

Embedded Systems

- Increasingly used in critical sectors
  - Defense, medical, power, …
- Malicious and accidental failures can have dire consequences
- Embedded systems are not “all hardware”
  - They have software too☺
- Autonomous nature

Dynamic and Configurable Environment

- Embedded systems are highly configurable
  - They have to work in many different scenarios
- Environment is highly dynamic
  - Think about embedded systems in a battlefield
  - Embedded system in a vehicle

Changing Functional Requirements

- Functional requirements of embedded systems change over time
- Embedded system deployed in a battlefield
  - Functional requirements change with mission

Interconnected Network of Components

- Embedded system are of a complex network of components
- Components might be hardware or software
- Source code might be available for some components
- COTS components (only binary available)
- Failure can create cascading events

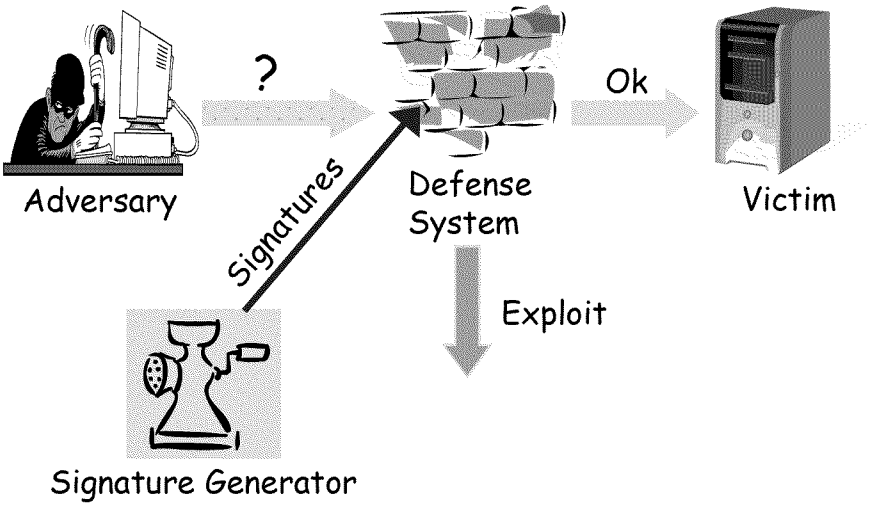
Recovery is Paramount

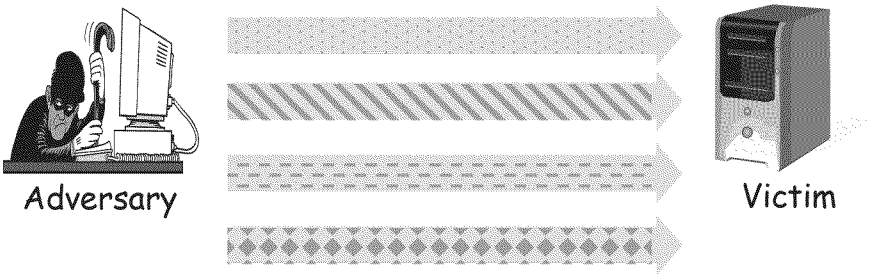
- Embedded systems used in critical applications
- In some cases recovery is paramount
- Recovery complicated by complex interaction of events
  - Failure can cause a complex cascade of events

Three Software Security Projects

- Automated generation of vulnerability signatures
- Retrofitting legacy code
- Static analysis of binaries
  - Malware Detection

Motivating Scenario for Automatic Signature Generation



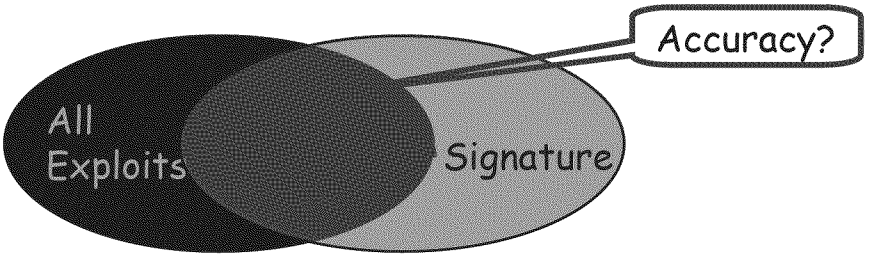


...

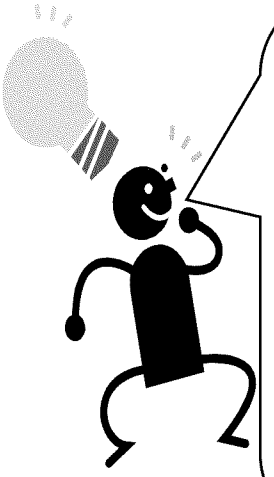
Many, perhaps infinite,
Polymorphic variants

Goals for Automatic Signature Generation

- Create signature that matches exploits
- Reason about signature accuracy
  - —Does it match legitimate traffic (false +)?
  - —Does it miss exploits (false -)?



Our Contribution: A Language-Centric Approach



- Focus on the language of the vulnerability
  - ➡ Reason about signature via language
  - ➡ Language captures all exploits
- New methods for Automatic vulnerability signature creation
  - ➡Opens doors to PL techniques

Language of a Particular Vulnerability

- A vulnerability is defined by:

1. What – The Vulnerability Condition:
   Necessary conditions to violate safety
2. Where – The Vulnerability Point:
   Location vulnerability condition first satisfied

The Vulnerability Language is all input strings reaching the vulnerability point meeting the vulnerability condition.

HTTP-like Running Example

```
1.  int check_http(char input[9])
2.  {
3.    if(strcmp(input, “get”,3) != 0 ||
4.        strcmp(input, “head”,4) != 0) return -1;
5.    if(input[4] != ‘/‘) return -1;
6.    int I = 5;
7.    while(input[I] != ‘\n‘){  I++; }
8.    input[I] = 0;
9.    return I;
10. }
```

Our implementation is on binaries

Example Input: get_/aaaa\n

```
1.  int check_http(char input[9])
2.  {
3.    if(strcmp(input, “get”,3) != 0 ||
4.        strcmp(input, “head”,4) != 0) return -1;
5.    if(input[4] != ‘/‘) return -1;
6.    int I = 5;
7.    while(input[I] != ‘\n‘){  I++; }
8.    input[I] = 0;
9.    return I;
10. }
```

Vulnerability Point

Example Input: get_/aaaa\n

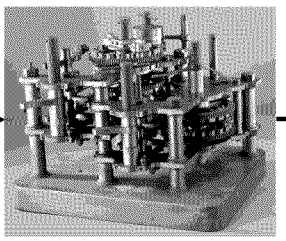
```
1.  int check_http(char input[9])
2.  {
3.    if(strcmp(input, “get”,3) != 0 ||
4.        strcmp(input, “head”,4) != 0) return -1;
5.    if(input[4] != ‘/‘) return -1;
6.    int I = 5;
7.    while(input[I] != ‘\n‘){  I++; }
8.    input[I] = 0;
9.    return I;
10. }
```

Vulnerability Condition: I >= 9

Retrofitting legacy code

Need systematic techniques to retrofit legacy code for security

Legacy code → **INSECURE**



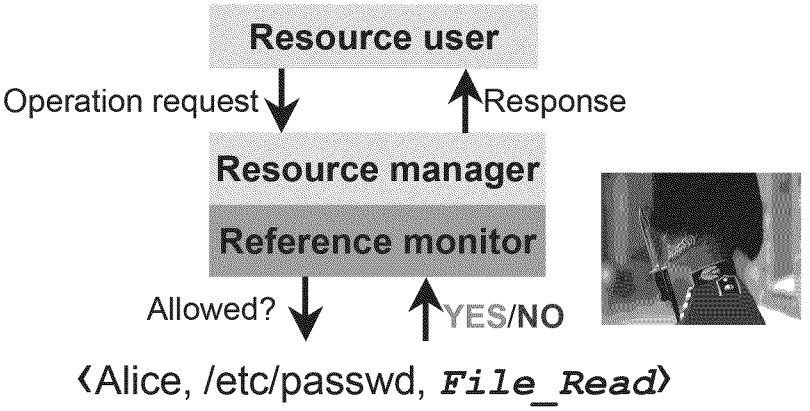
→ Retrofitted code **SECURE**

Retrofitting legacy code

Need systematic techniques to retrofit legacy code for security

- Enforcing type safety
  - CCured [Necula *et al.* '02]
- Partitioning for privilege separation
  - PrivTrans [Brumley and Song, '04]
- Enforcing authorization policies

Enforcing authorization policies

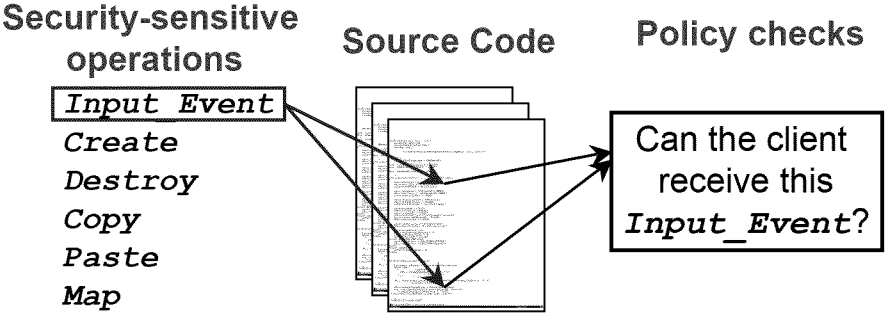


Retrofitting for authorization

- Mandatory access control for Linux
  - Linux Security Modules [Wright *et al.*,'02]
  - SELinux [Loscocco and Smalley,'01]
- **Painstaking, manual procedure**
  - Trusted X, Compartmented-mode workstation, X11/SELinux [Epstein *et al.*,'90][Berger *et al.*,'90][Kilpatrick *et al.*,'03]
- Java Virtual Machine/SELinux [Fletcher,'06]
- IBM Websphere/SELinux [Hocking *et al.*,'06]

Retrofitting lifecycle

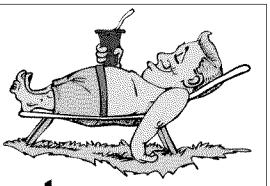
1. Identify security-sensitive operations
2. Locate where they are performed in code
3. Instrument these locations



Problems

- Time-consuming
  - X11/SELinux ~ 2 years [Kilpatrick *et al.*, '03]
  - Linux Security Modules ~ 2 years [Wright *et al.*, '02]
- Error-prone [Zhang *et al.*, '02][Jaeger *et al.*, '04]
  - Violation of complete mediation
  - Time-of-check to Time-of-use bugs

Our approach



Reduces manual effort

- Retrofitting takes just a few hours
  - Automatic analysis: ~ minutes
  - Interpreting results: ~ hours

Reduces errors

- Basis to prove security of retrofitted code

Malspec: *Self-Propagation by Email*

```
push    10h
push    eax
push    edi
call    connect
push    esi
push    eax
push    [ebp+hMem]
call    wsprintfA
add     esp, 0Ch
push    [ebp+hMem]
call    lstrlenA
push    0
push    eax
push    [ebp+hMem]
push    ebx
push    eax
push    ecx
push    edi
call    send
```

Netsky.B

=

Connect

Syntactic component describes temporal constraints.

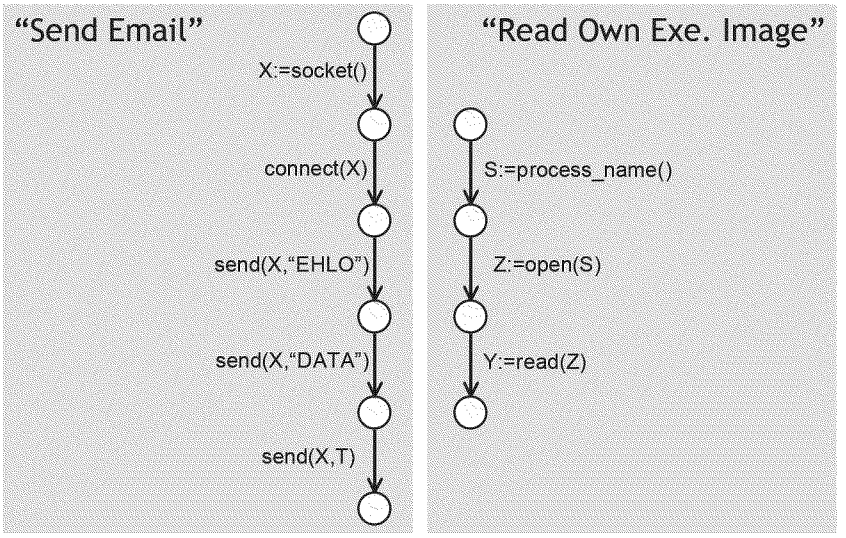
+

Semantic component describes dependency constraints.

Send

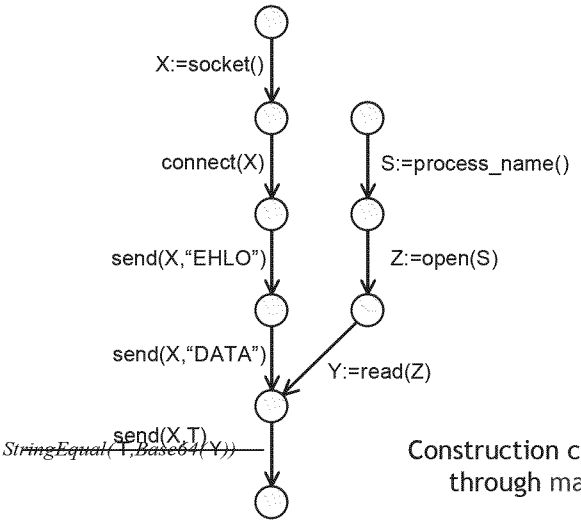
$Arg_1 = X$
&
Arg_2 = "EHLO.*"

Building a Real Malspec



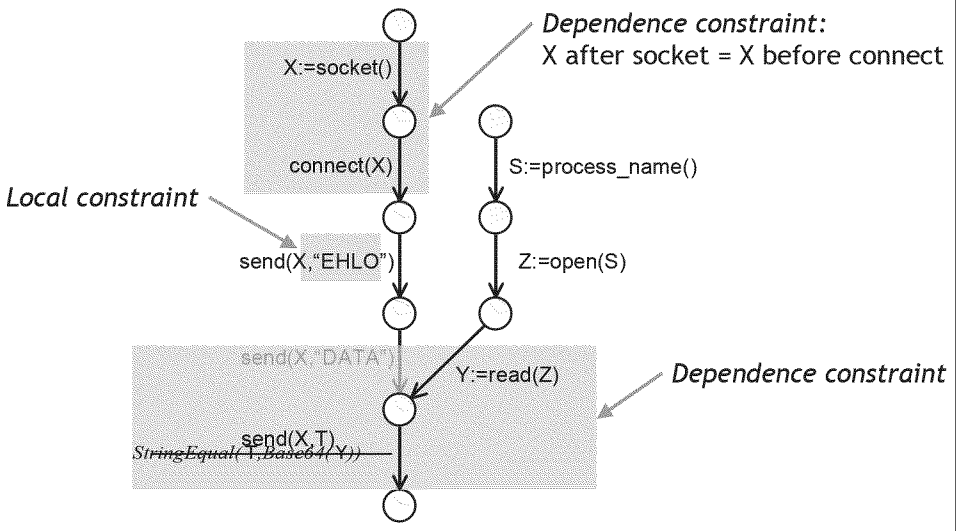
Malspec: *Self-Propagation by Email*

AND-OR graph

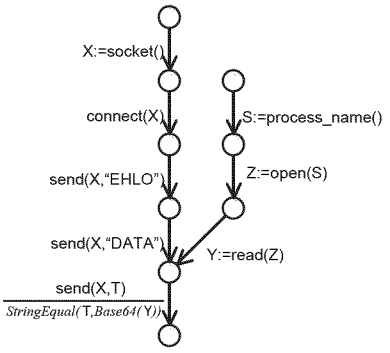


Construction can be automated through malspec mining.

Malspec Constraints



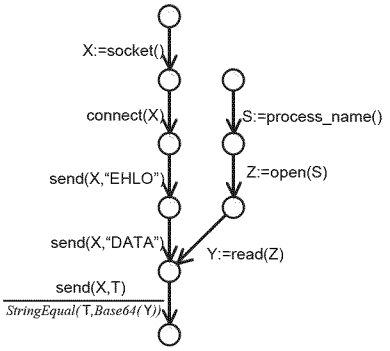
Malspecs Benefits



- ✓ Symbolic variables
- ✓ Constraint-based execution order
- ✓ Independent of obfuscation artifacts

Expressive to describe even obfuscated behavior.

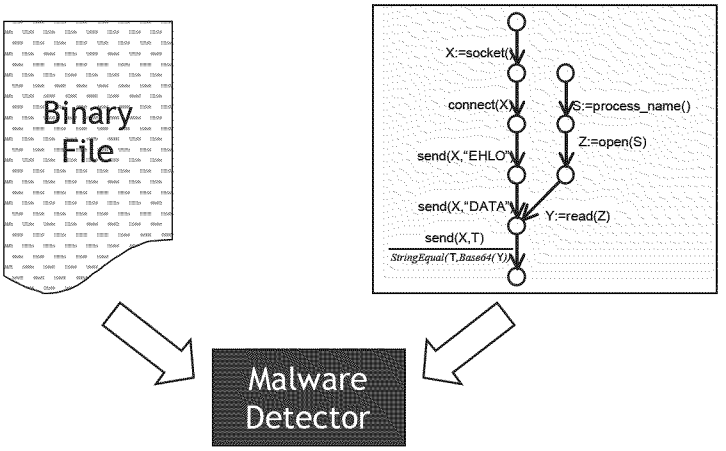
Malspec Detection Strategies



- Static analysis
- Dynamic analysis
  - Host-based IDS
  - Inline Reference Monitors

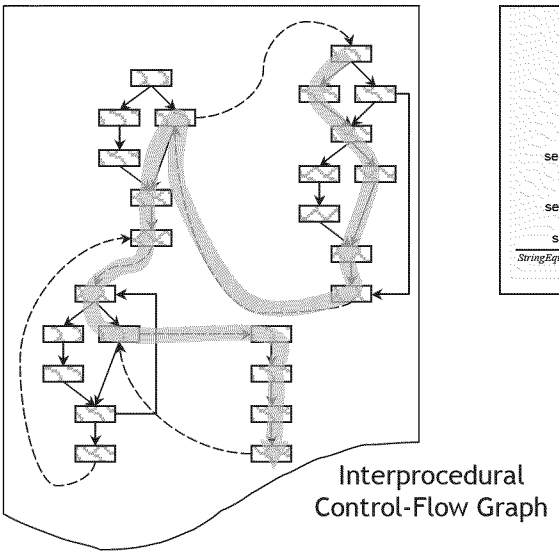
Malspecs are independent of detection method.

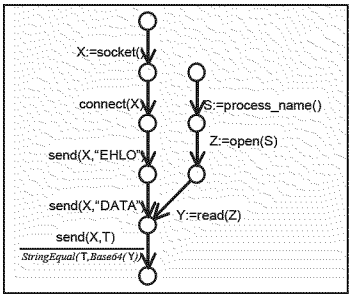
Detection of Malicious Behavior



Goal: Find a program path that matches the malspec.

Find A Malicious Program Path





Stable Environment Assumption

- All the above mentioned work assumes a "nearly" stable environment
- Example: web server
  - Is configurable, but the environment is not that rich
  - Environment is not too dynamic
  - Not rich interaction with other components
- Incorporating "dynamic environments" into the techniques described before is a challenge

Vulnerability Assessment in Presence of a Dynamic Environment

- Dynamic and static analysis techniques assume a relatively stable environment
- Parameterized static analysis
  - Parameterize static analysis with environment assumptions
  - Similar to assume-guarantee reasoning in model checking
- Parameterized vulnerability signatures

Recovery from Failures

- A failure (malicious or benign) can cause a complex cascade of events
- Need to understand the complex cascade of events caused by a failure
- Need to analyze the complex network in components in totality
  - Scalability
  - Compositional analysis

Questions

- My web page
  - http://www.cs.wisc.edu/~jha